

Algorithm efficiency and multidimensional arrays

Searching, sorting and the cost of scale
Arrays, rows, columns and nested loops

Prof. Dr. Mehmet Göktürk · Gebze Technical University

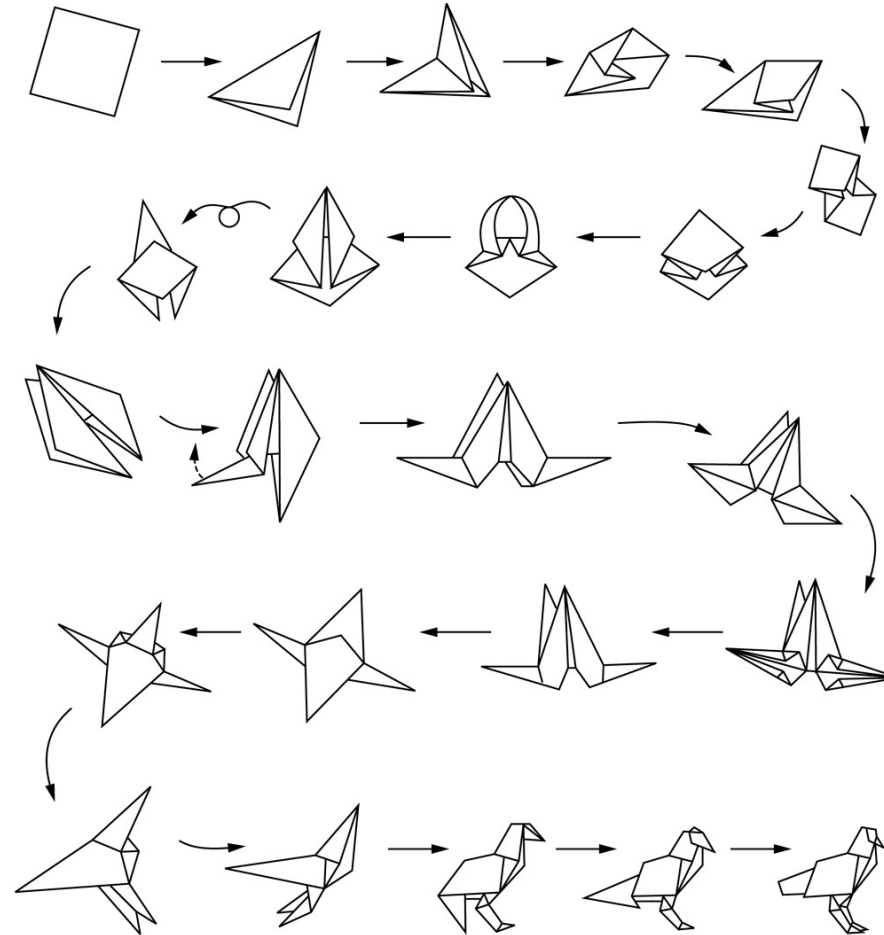
The questions for this week

How can two correct algorithms solve the same problem with very different amounts of work?

What assumptions make a faster method possible?

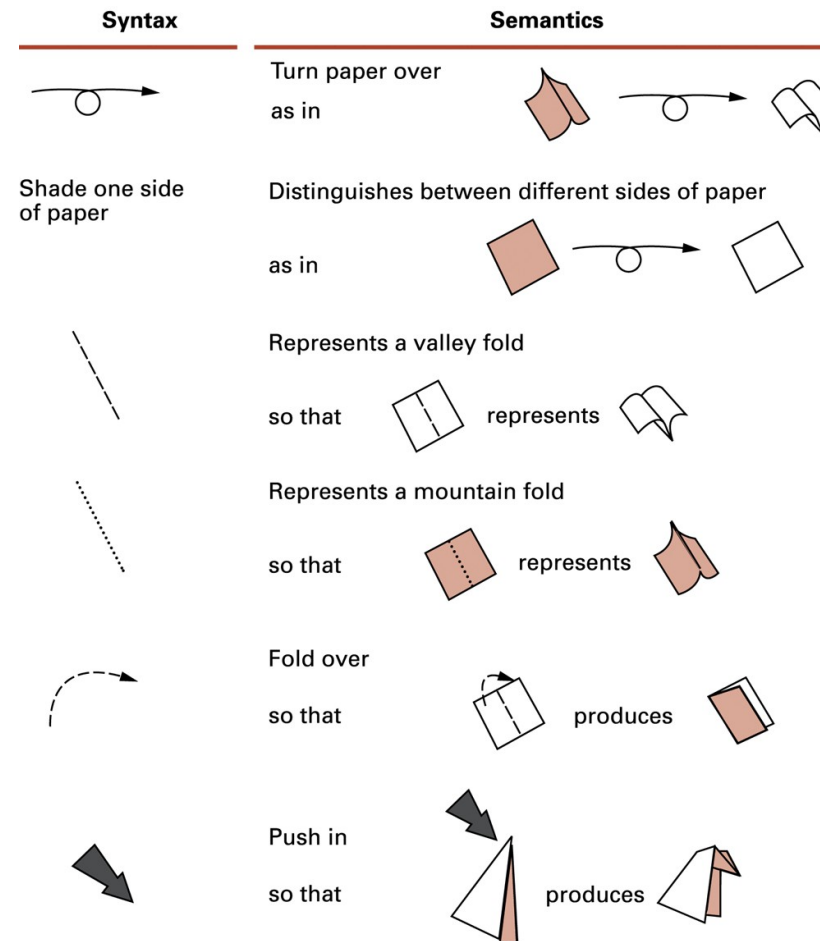
How do indices and nested loops organize a collection of readings?

An algorithm can be visual



A sequence is useful only when the reader can interpret each operation.

Primitives need agreed meanings



A small vocabulary of precise operations makes a long visual description executable.

Problem, algorithm and program

Problem: find whether a target appears in a collection.

Algorithm: inspect items in a defined order, with a rule for stopping.

Program: express that procedure in a particular language. Correctness and cost concern the algorithm as well as its implementation.

Problem solving before implementation

Understand the input, the required result and the constraints.

Develop a plan using examples, a simpler problem or a useful representation.

Trace the plan, including failure cases. Then implement and evaluate it. These are Pólya's problem-solving stages.

Search has a contract

Input: n elements and a target. For our examples, n is nonnegative and the array contains at least n accessible elements.

Output: an index containing the target, or -1 if it is absent. Duplicate matches may return any matching index unless the contract says otherwise.

An empty collection must report “not found” without reading an element.

Sequential search: a worked trace

Index	Value	Target = 31
0	18	No match
1	7	No match
2	31	Match; stop
3	12	Not examined

Order is unnecessary. Each probe tests one element against the target.

Sequential search and absence

Inspect each element until a match appears or the collection ends.

Target 9 in [18, 7, 31, 12]: all four elements must be examined.

After examining a prefix without success, none of those examined elements equals the target. This is the fact preserved by the loop.

Counting work, not stopwatch time

Situation in n elements	Element probes
First element matches	1
Last element matches	n
Target absent	n
Uniformly likely successful position	$(n + 1) / 2$ on average

The average requires this probability assumption. Empty input makes zero probes.

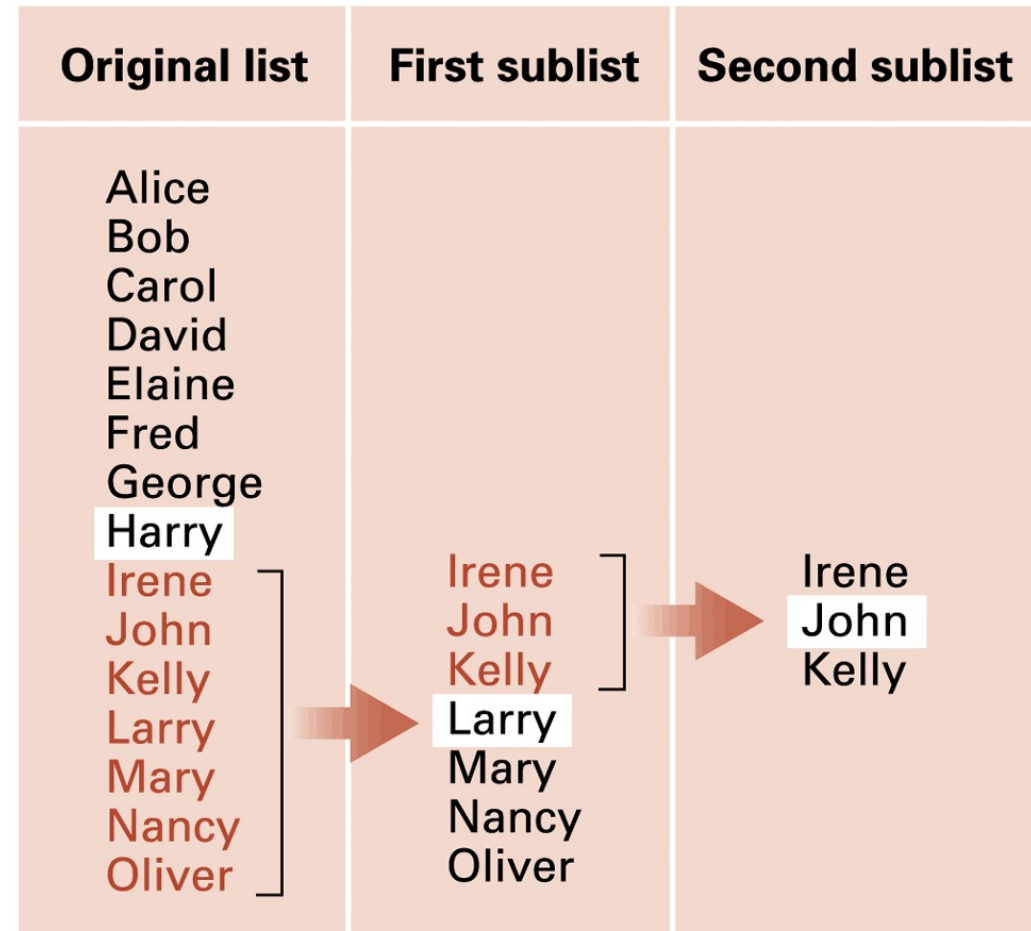
Binary search needs order

Keep the candidate interval in a sorted array. Inspect its middle element.

If the target is smaller, eliminate the middle and everything to its right. If larger, eliminate the middle and everything to its left.

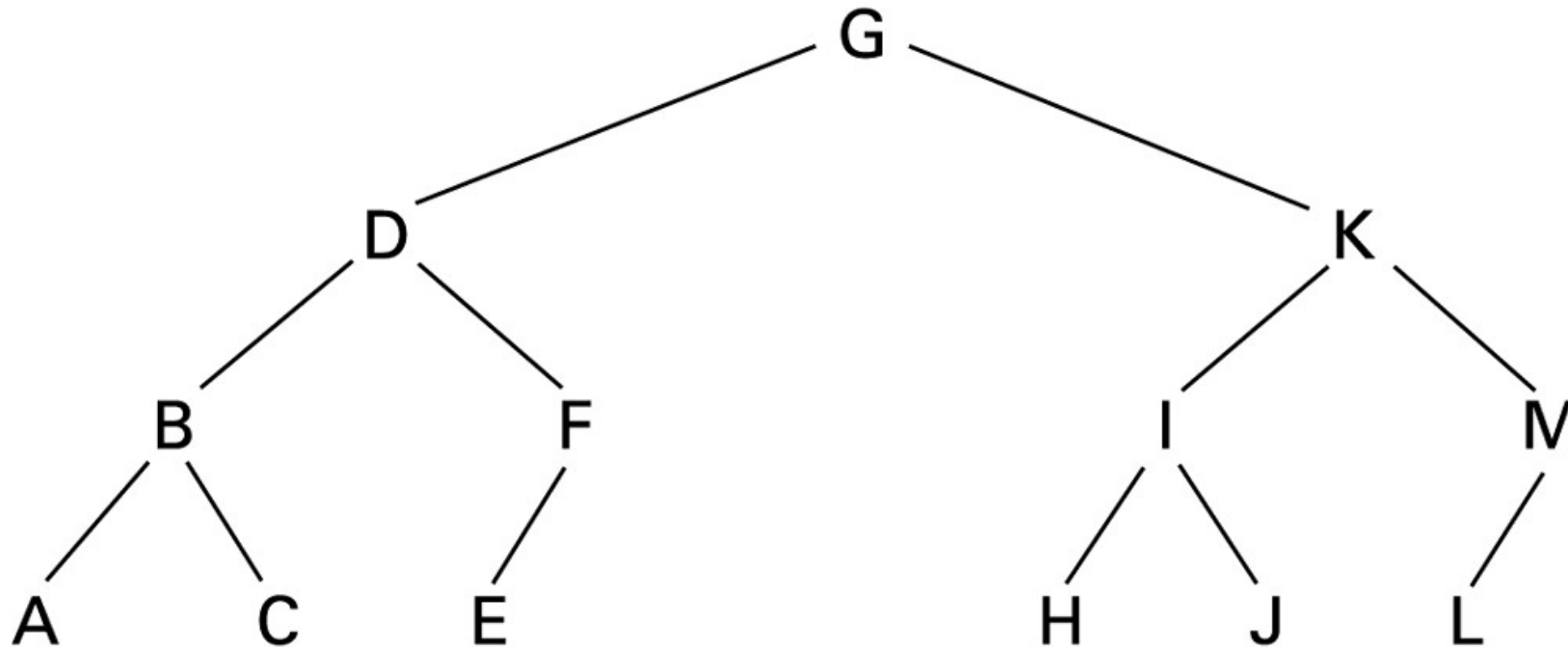
Discarding half is justified by the ordering. It is unsafe on an arbitrary unsorted array.

Binary search in the original example



The search for John narrows the candidate list until its middle entry is John.

An ordered tree makes the search visible



Find J

Tree: $G \rightarrow K \rightarrow I \rightarrow J$

4 node checks

Scan A, B, ..., J

10 item checks

Smaller values: left

Larger values: right

The other branch is skipped.

A shallow tree gives short paths. An unbalanced tree can require a linear search.

Binary search on a sorted array uses midpoint indices; it does not need stored tree links.

Binary search: index trace

Candidate indices	Middle value	Target = 31
0...6	$a[3] = 18$	Keep 4...6
4...6	$a[5] = 31$	Found at index 5

Sorted data: [3, 7, 12, 18, 24, 31, 45]. A probe is one middle-element inspection.

A failed binary search

Candidate indices	Middle value	Target = 30
0...6	18	Keep 4...6
4...6	31	Keep 4...4
4...4	24	Keep 5...4: empty

An empty interval is a normal result, not an invitation to read another element.

Why binary search finishes

If the target exists, it stays inside the remaining candidate interval.

Every unsuccessful probe reduces the candidate interval to at most half its previous size.

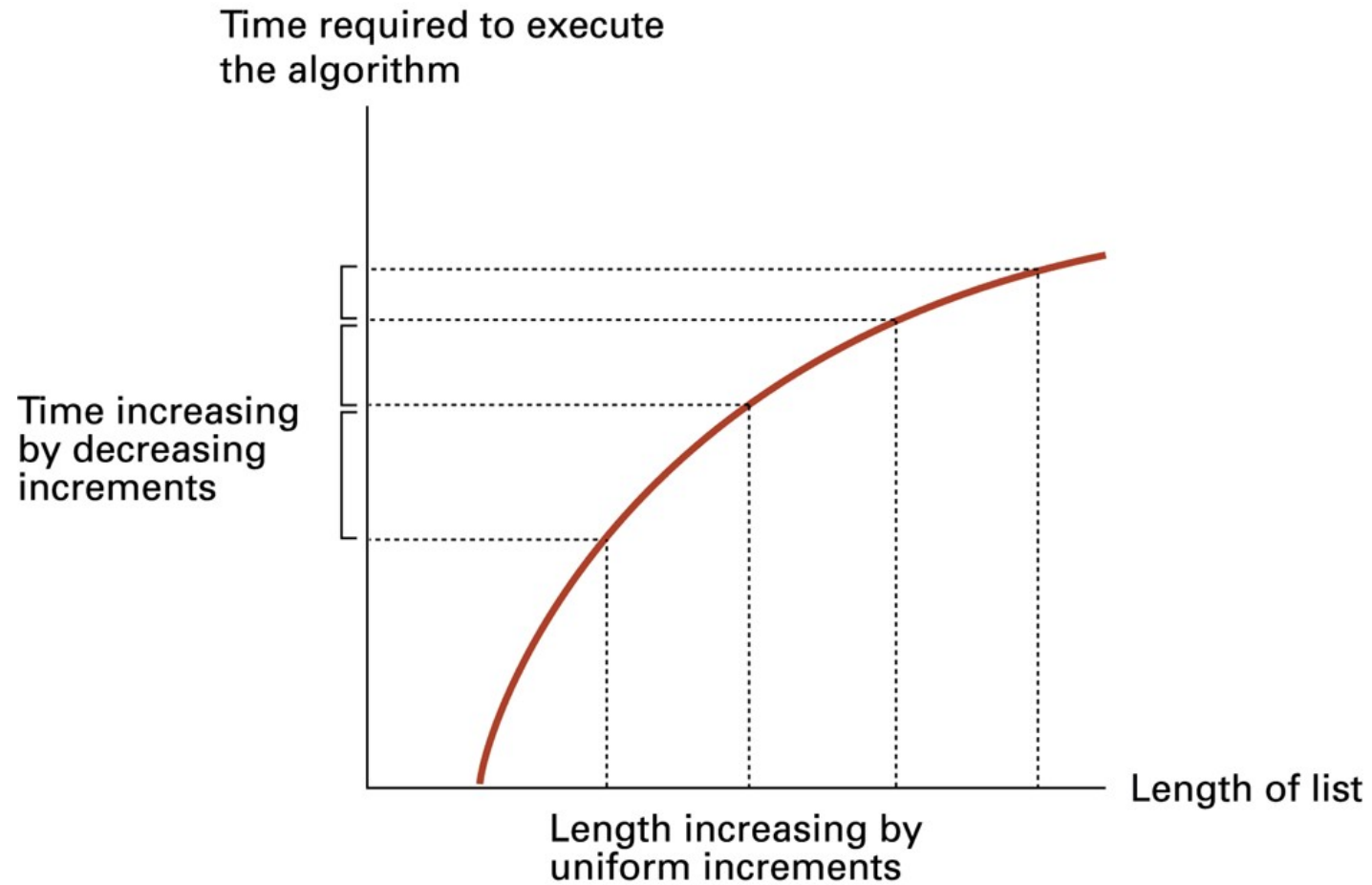
Eventually the interval contains a matching element or becomes empty. Correctness needs both the preserved fact and progress toward termination.

Input size changes the search cost

n	Sequential worst case	Binary worst case
15	15 probes	4 probes
255	255 probes	8 probes
1,023	1,023 probes	10 probes
1,048,575	1,048,575 probes	20 probes

For $n = 2^k - 1$, at most k middle-element probes. Sorting cost is not included.

Logarithmic growth



The original curve shows diminishing increases. The model counts work; measured time also depends on implementation.

Sorting has a cost too

For one query in unsorted data, a linear scan may be the simpler choice.

For many queries in mostly unchanged data, sorting once can make repeated searches cheaper.

Include preparation, updates and memory use when choosing a method. “Binary search is faster” needs a stated setting.

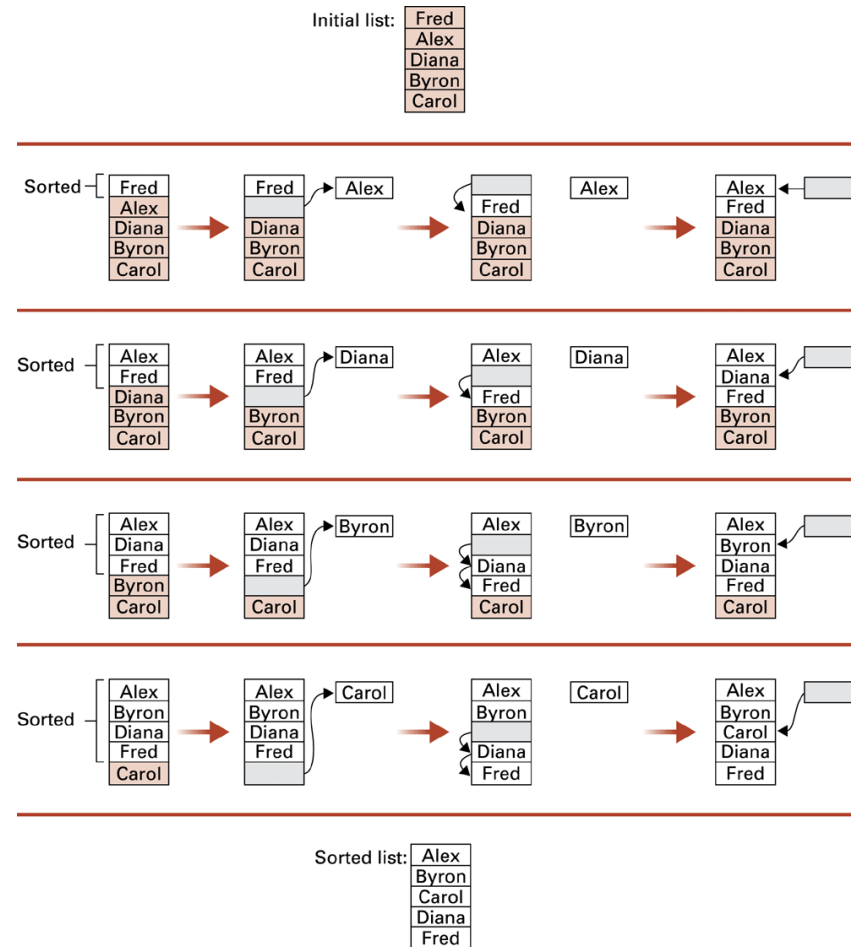
Sorting: what must stay true?

Reorder the input so that adjacent entries follow the chosen order.

The result must contain the same entries, including repeated values. Losing an entry can produce an ordered but incorrect result.

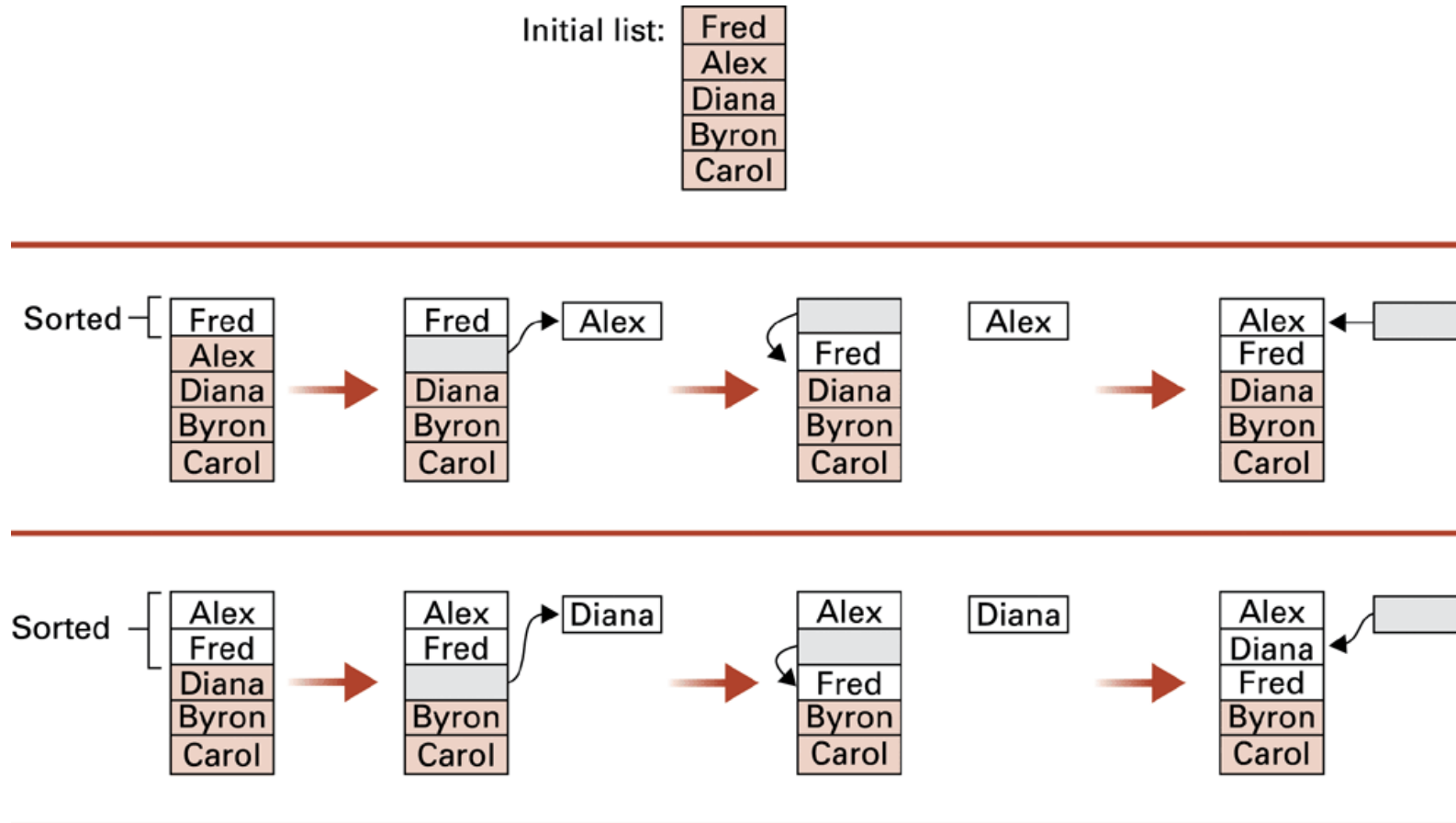
For records, decide which field supplies the key. Sorting only one column can break the relationship between fields.

Insertion sort: the original sequence



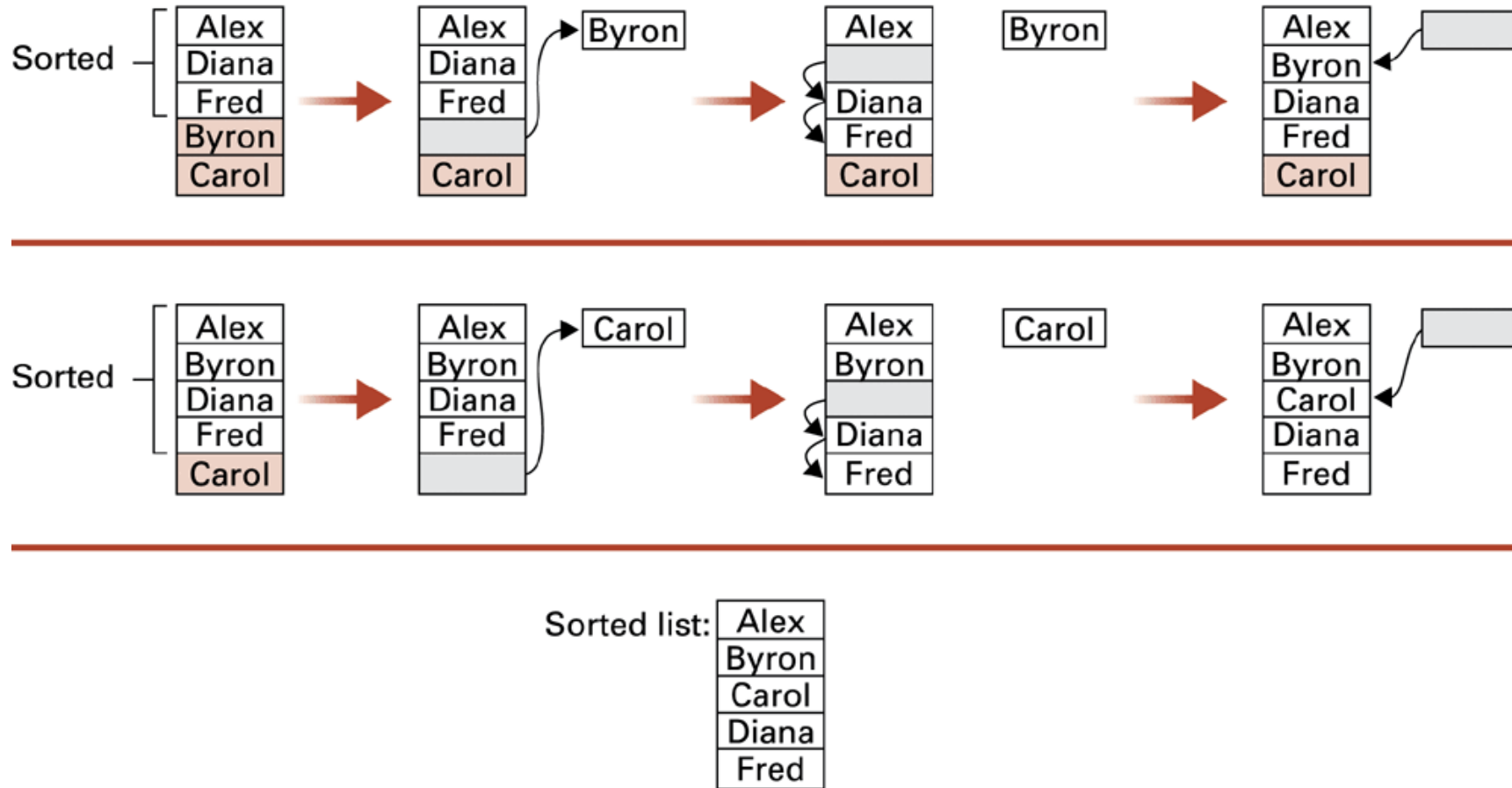
A sorted prefix grows by inserting the next entry into its proper place. Larger detail views follow.

Insertion sort: first insertions



Alex moves before Fred. Diana then fits between Alex and Fred.

Insertion sort: completing the list



Byron and Carol are inserted into the already sorted prefix.

Insertion sort with repeated values

Pass	Values	Sorted prefix
Initial	7, 3, 5, 3	7
Insert 3	3, 7, 5, 3	3, 7
Insert 5	3, 5, 7, 3	3, 5, 7
Insert final 3	3, 3, 5, 7	All four values

Move values greater than the key. This preserves the order of equal keys in this version.

Insertion sort in the worst case

Comparisons made for each pivot

Initial list	1st pivot	2nd pivot	3rd pivot	4th pivot	Sorted list
Elaine David Carol Barbara Alfred	1 → Elaine David Carol Barbara Alfred	3 → David 2 → Elaine Carol Barbara Alfred	6 → Carol 5 → David 4 → Elaine Barbara Alfred	10 → Barbara 9 → Carol 8 → David 7 → Elaine Alfred	Alfred Barbara Carol David Elaine

Reverse order makes each new pivot cross the entire sorted prefix: $1 + 2 + 3 + 4 = 10$ comparisons.

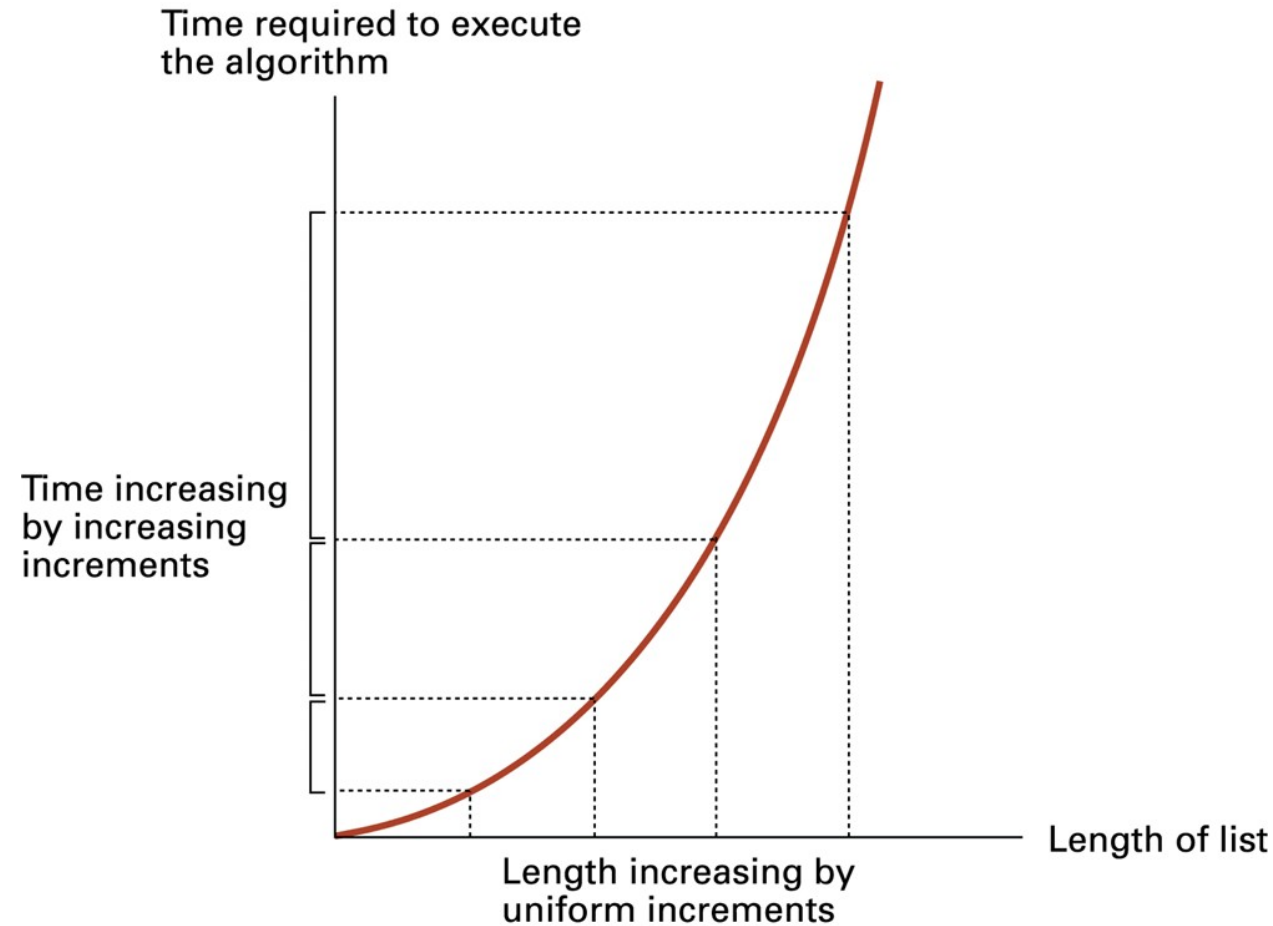
A quadratic amount of work

For reverse-sorted input, insertion sort makes $1 + 2 + \dots + (n - 1)$ key comparisons.

The sum is $n(n - 1)/2$. At $n = 100$, that is 4,950 comparisons; at $n = 1,000$, it is 499,500.

Already sorted input is different: the usual implementation makes $n - 1$ key comparisons and no shifts.

Quadratic growth



Doubling a large input roughly quadruples the leading quadratic work. This is a growth model, not a timing measurement.

Growth classes describe scaling

Class	Example work	Doubling n
$\Theta(1)$	Read one known array element	No size-dependent increase
$\Theta(\log n)$	Binary search, worst case	About one more probe
$\Theta(n)$	Sequential search, worst case	About twice the work
$\Theta(n^2)$	Insertion sort, worst case	About four times the work

Θ describes a tight growth class. O gives an asymptotic upper bound. Always name the case.

A faster machine or a better algorithm?

Suppose machine A makes 1 million model operations per second, and B makes 100 million.

For $n = 100,000$: a quadratic n^2 model takes 100 seconds on B; a linear n model takes 0.1 seconds on A.

This is an illustrative model, not a benchmark. Growth can outweigh a large hardware advantage.

Loops do not determine cost by appearance

Two consecutive full-array loops do $n + n = 2n$ iterations, so growth is linear.

A loop over r rows containing a loop over c columns visits $r \times c$ entries. For a square $n \times n$ table, that is n^2 .

A loop that halves its candidate interval has logarithmic iterations. Inspect the bounds and how they change.

An auxiliary structure may use extra memory to avoid repeated computation.

In-place insertion sort uses a constant amount of auxiliary storage. Binary search can also use a fixed set of index variables.

A recursive version keeps call state. Algorithm choice includes memory, predictability and the properties of the input.

Use empty input, one item, repeated values, missing targets and targets at both ends.

For sorting, test both order and preservation of the input values.

Tests reveal counterexamples. An invariant explains why every iteration preserves a required property.

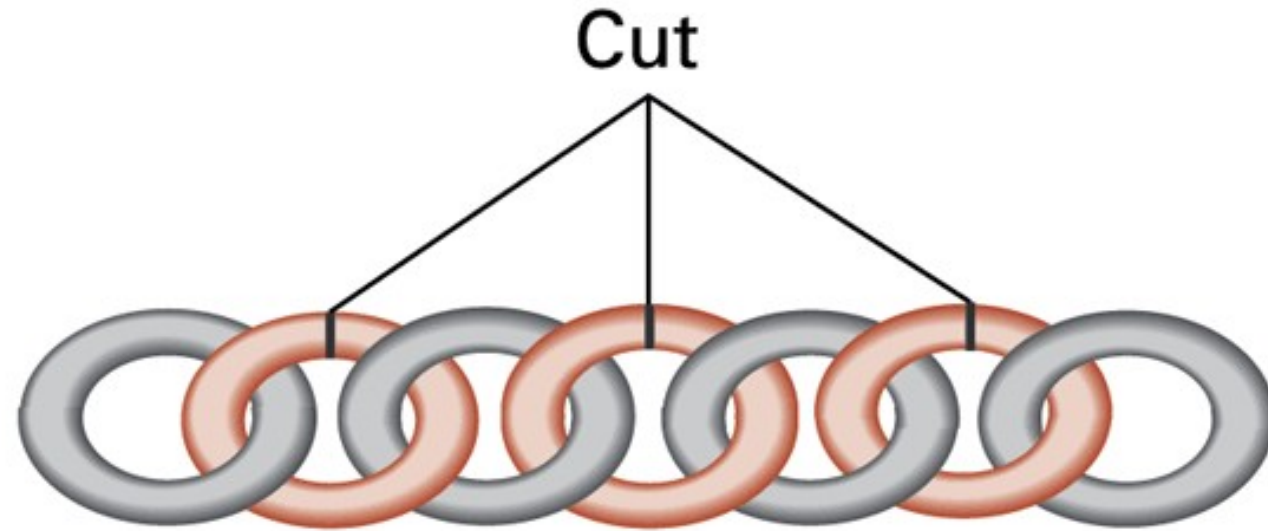
A problem-solving challenge

A hotel charges one gold-chain link per night for seven nights. Payment is settled each night, and the hotel can return earlier pieces as change.

You may cut links to make reusable pieces. Minimize the number of links cut.

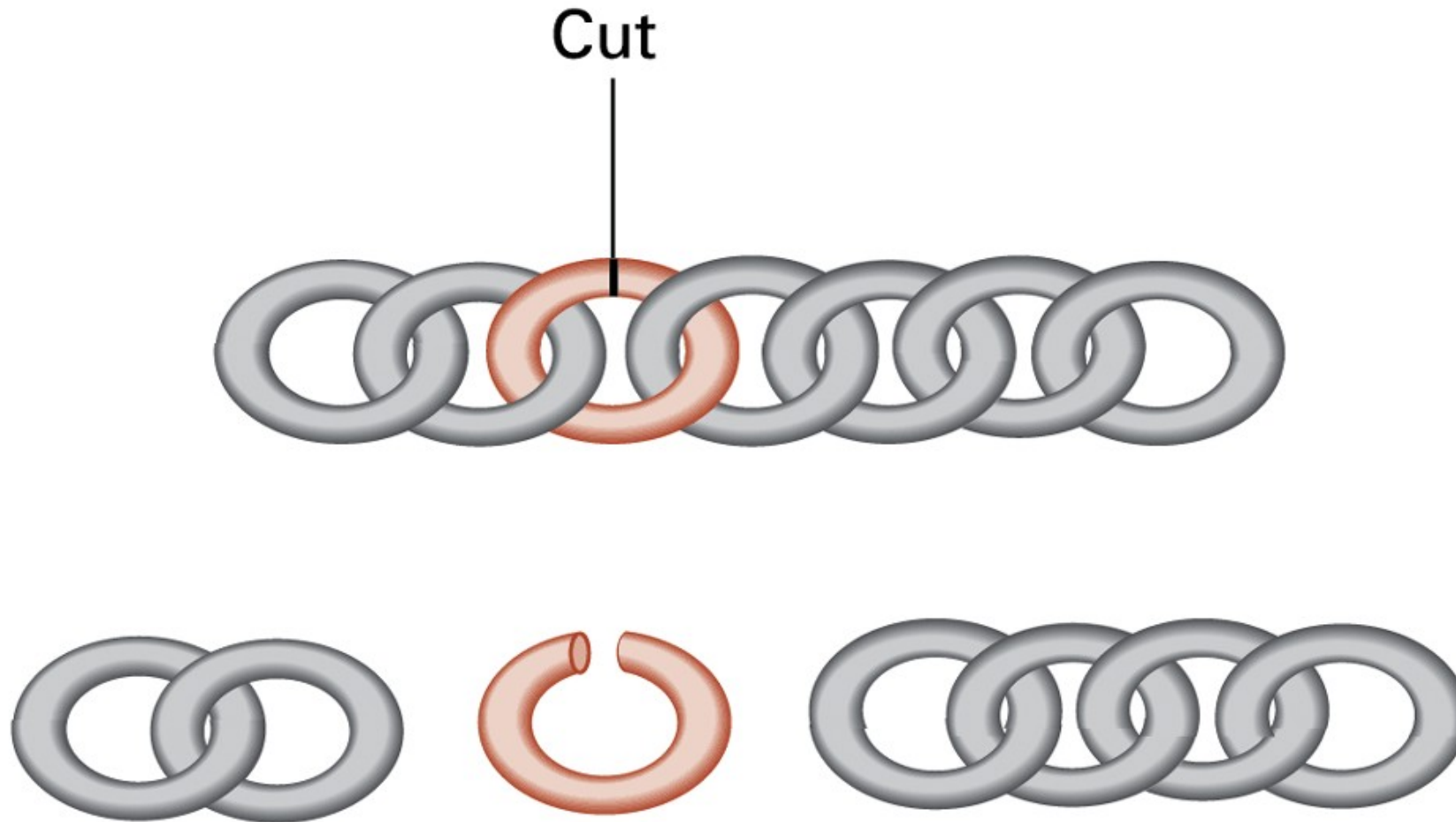
Before choosing cuts, list the totals that the pieces must represent: 1 through 7.

A solution with three cuts



Three cut links allow all seven links to be separated. This works, but can we do better?

A solution with one cut



Cutting the third link creates pieces worth 1, 2 and 4 links. They can represent every total from 1 to 7.

Payment by exchanging pieces

Night	Pieces held by hotel	Total
1	1	1
2	2	2
3	$2 + 1$	3
4	4	4
5	$4 + 1$	5
6	$4 + 2$	6
7	$4 + 2 + 1$	7

Change is allowed. Without that assumption, this solution does not apply.

The theory to carry forward

A faster method often exploits a useful property, such as sorted input.

Define the counted operation, input size and case before comparing efficiency.

Correctness, memory and preparation cost matter alongside the growth rate.

Organizing batches of readings

One-dimensional review, rows and columns
Tracing and reasoning before implementation

A fixed batch of readings

Index	0	1	2	3
Value	120	400	650	230

Four illustrative ADC counts on a 0–1023 scale. Their sum is 1,400; their mean is 350.

A traversal has a clear boundary

Initialize the total to zero, visit indices 0 through $n - 1$, and add each value exactly once.

After i visits, the total equals the sum of elements with indices below i .

For $n = 0$ the sum is zero. A mean needs a separate empty-input decision because division by zero is invalid.

One traversal in C

```
int sum(const int a[], int n) {  
    int total = 0;  
    for (int i = 0; i < n; ++i)  
        total += a[i];  
    return total;  
}
```

Contract: $0 \leq n \leq 12$; at least n accessible elements.

Values are 0–1023. Total is at most 12,276.

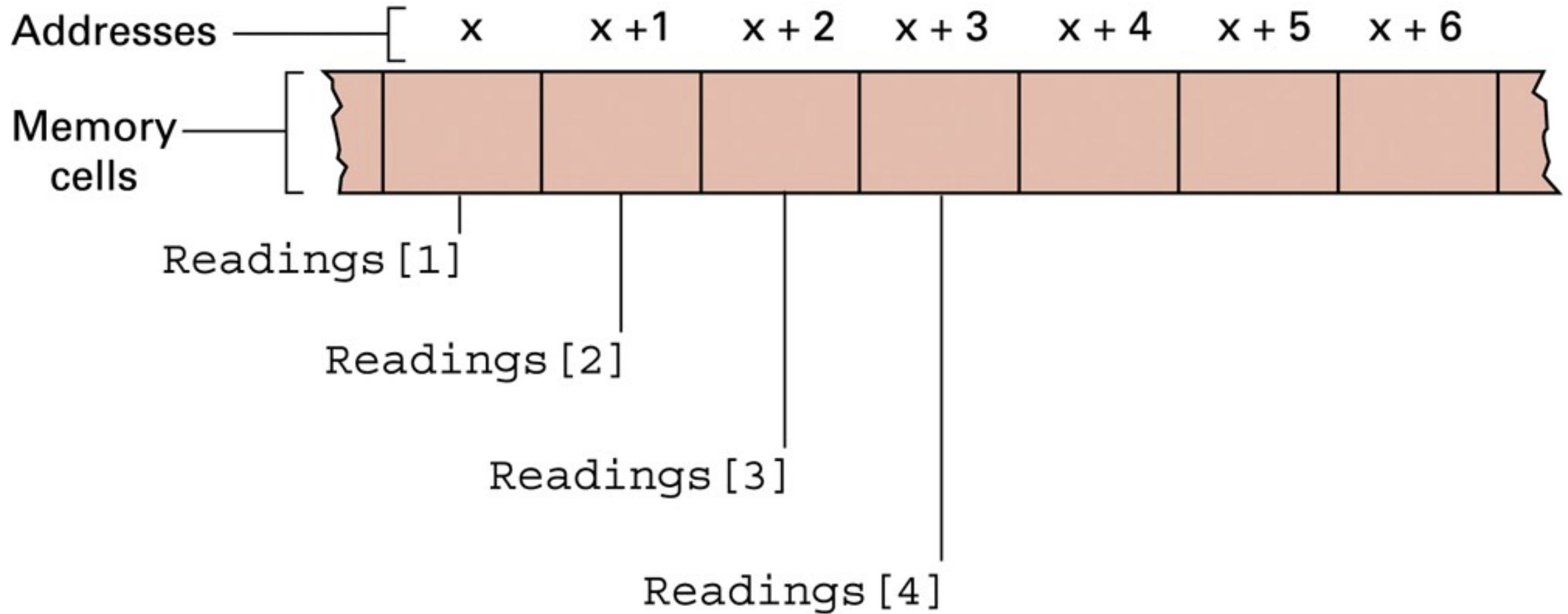
This bound fits even an implementation with a 16-bit int.

Tracing the total

i before visit	Value added	Total afterward
0	120	120
1	400	520
2	650	1,170
3	230	1,400

At exit $i = 4$, so the body does not access $a[4]$.

Array storage and indexing



The drawing uses one-based labels and abstract cells. C starts at index 0; byte spacing is `sizeof(element)`.

Three batches form a table

Batch / sample	0	1	2	3
0	120	400	650	230
1	300	310	290	300
2	800	100	900	200

Rows are batches; columns are successive samples within each batch. No matrix algebra is needed.

Declaring a two-dimensional array

```
int readings[3][4] = {  
    {120, 400, 650, 230},  
    {300, 310, 290, 300},  
    {800, 100, 900, 200}  
};
```

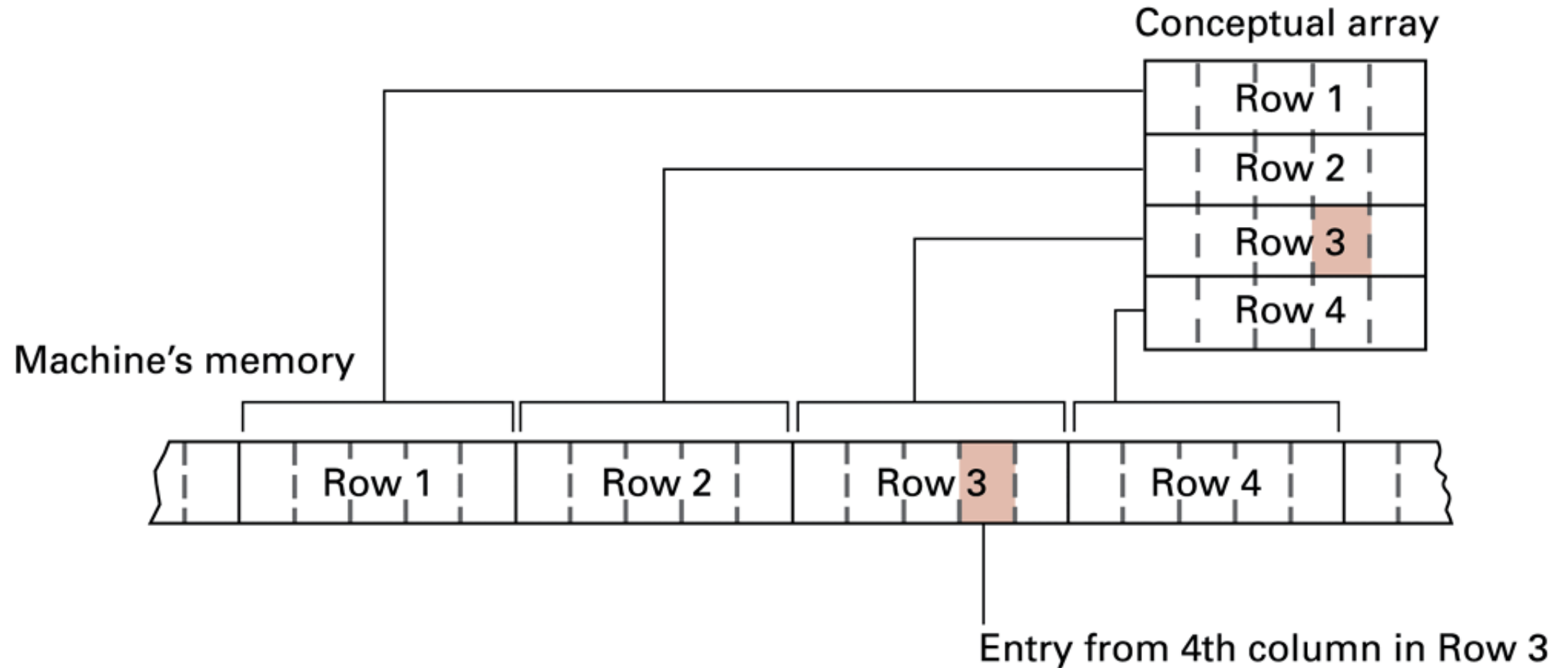
```
int value = readings[2][1];
```

Three rows, four elements
per row.

Valid indices: row 0–2,
column 0–3.

readings[2][1] is 100.

Rows laid out in memory



The highlighted fourth column of Row 3 corresponds to C index [2][3] in this four-by-five example.

C stores a true two-dimensional array as consecutive row arrays. Each row contains consecutive elements.

For `int readings[3][4]`, element `[r][c]` is $r \times 4 + c$ elements from the start, for valid indices.

Element `[2][1]` is 9 elements from the start. Its byte offset is $9 \times \text{sizeof(int)}$; an `int` is not universally four bytes.

The nested-loop visit order

Outer row	Inner columns	Elements visited
0	0, 1, 2, 3	120, 400, 650, 230
1	0, 1, 2, 3	300, 310, 290, 300
2	0, 1, 2, 3	800, 100, 900, 200

The inner loop starts again at column 0 for each row. Total visits = $3 \times 4 = 12$.

One total per row

```
for (int r = 0; r < 3; ++r) {  
    int total = 0;  
    for (int c = 0; c < 4; ++c)  
        total += readings[r][c];  
    printf("%d\n", total);  
}
```

Reset total inside the outer loop.

Each inner loop accumulates one row.

Output: 1400, 1200, 2000.

Row totals and means

Batch	Sum	Mean
0	1,400	350.0
1	1,200	300.0
2	2,000	500.0

Use total / 4.0 for a floating-point mean. An all-integer division may discard a fractional part.

Accumulator placement changes meaning

A total initialized before both loops accumulates all twelve entries: 4,600.

A total reset once per row produces three separate totals.

A total reset inside the inner loop retains only the current value. Trace where state is initialized and updated.

A column asks a different question

Sample position	Across batches	Column total
0	120, 300, 800	1,220
1	400, 310, 100	810
2	650, 290, 900	1,840
3	230, 300, 200	730

A column compares the same position across batches. It is meaningful only if the batches align.

Bounds apply to every dimension

`readings[2][3]` is valid. `readings[3][0]` and `readings[0][4]` are out of bounds.

Contiguous storage does not make an out-of-range subscript valid. C does not automatically check array bounds.

Use the row count for the outer bound and the column count for the inner bound.

Reading order is part of the data

The rows preserve capture order. Sorting readings in place changes that order.

Sort a copy if later processing still needs the time sequence.

The same twelve numbers can support different questions, but interpretation depends on what each index represents.

Work first with saved batches so the expected answers are repeatable.

Capture and transport helpers can supply Arduino data to the PC. The array reasoning remains the same.

The lab focuses on a 1D traversal and a small 2D adaptation. Sorting benchmarks and a third-dimension implementation are outside its planned scope.

Check your understanding

Question	Answer
Why can binary search discard half?	Sorted order rules out that half.
What is the insertion-sort worst case?	Reverse order; $\Theta(n^2)$ work.
How many visits in a 3×4 traversal?	12.
Why reset a row total inside the outer loop?	Each row needs a fresh accumulator.

Explain the reason before looking at the answer.

Brookshear & Brylow, Computer Science: An Overview, 12th Global Edition: Chapter 5 on algorithms and Chapter 8 on data abstraction.

Original instructor Sets 7 and 8: algorithm traces, growth diagrams and array-storage figures.

Carnegie Mellon 15-122 binary-search notes; GNU C manual, “Multidimensional Arrays”. Full links are in the slide notes.

Worked examples and retained figures

Additional study material

The lecture does not require implementing every example

The ages-of-children puzzle

Three children have positive integer ages whose product is 36. Their sum matches a known house number.

Even after seeing the house number, the questioner needs another clue. The final clue says there is an oldest child.

Find the age triples, locate the ambiguous sum, then use the unique-oldest clue.

The original ages table

a. Triples whose product is 36

(1,1,36)	(1,6,6)
(1,2,18)	(2,2,9)
(1,3,12)	(2,3,6)
(1,4,9)	(3,3,4)

b. Sums of triples from part (a)

$$1 + 1 + 36 = 38$$

$$1 + 2 + 18 = 21$$

$$1 + 3 + 12 = 16$$

$$1 + 4 + 9 = 14$$

$$1 + 6 + 6 = 13$$

$$2 + 2 + 9 = 13$$

$$2 + 3 + 6 = 11$$

$$3 + 3 + 4 = 10$$

The ambiguous sum is 13: (1, 6, 6) and (2, 2, 9). A unique oldest child selects (2, 2, 9).

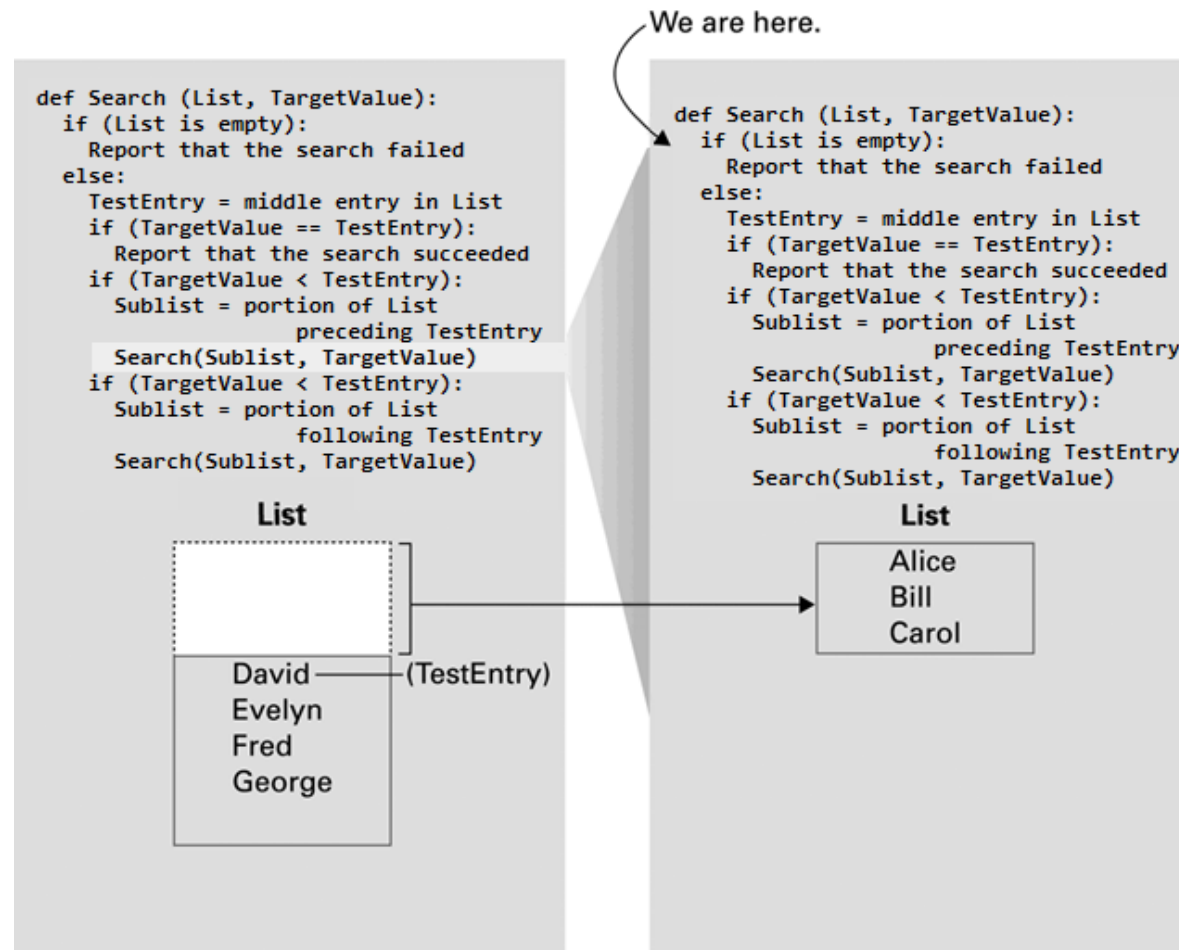
Recursion as an alternative description

A recursive procedure solves a smaller instance by calling itself. A base case finishes without another call.

Binary search can use recursion or an iterative interval. Halving the candidate set drives logarithmic search in both versions.

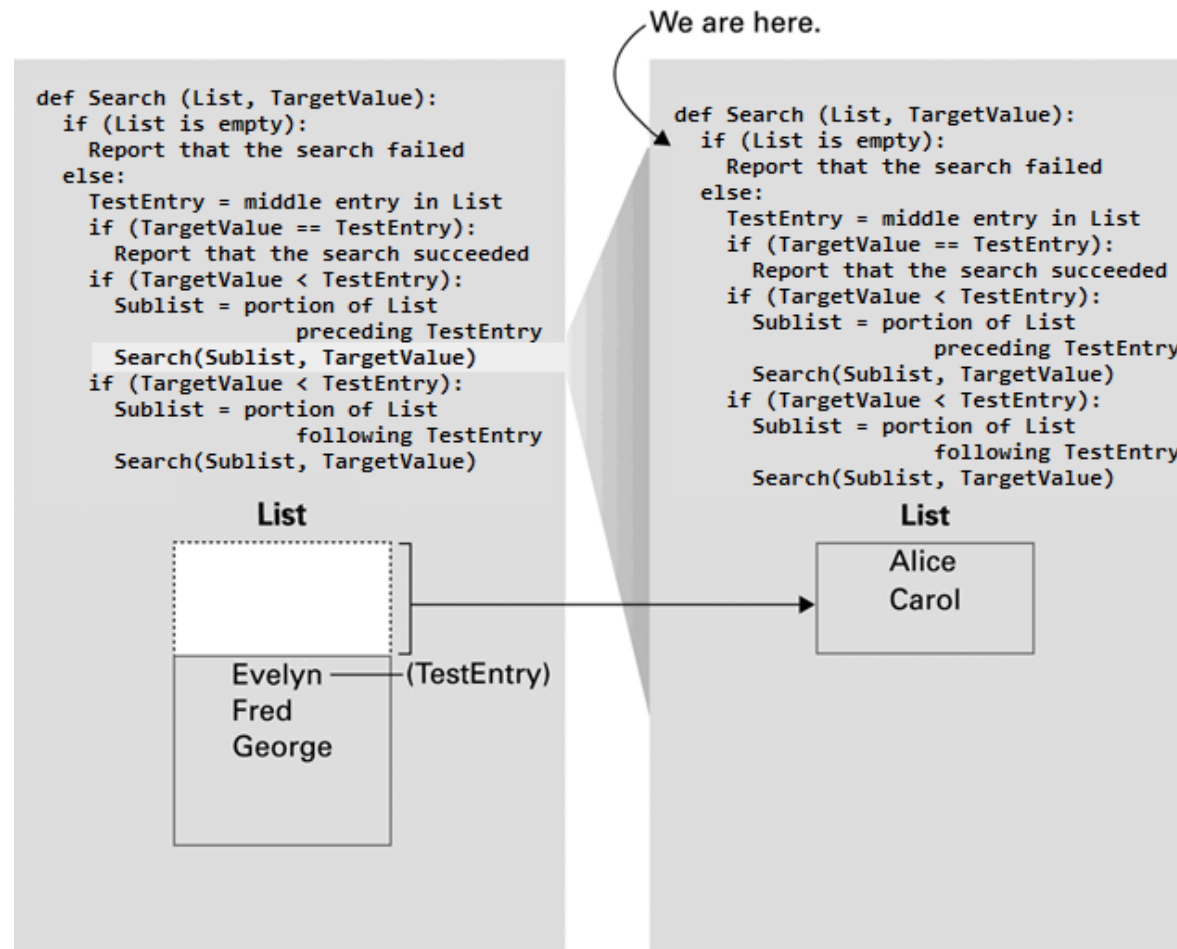
The following original snapshots show nested calls. They are a tracing reference, not an additional implementation task.

Recursive search: first snapshot



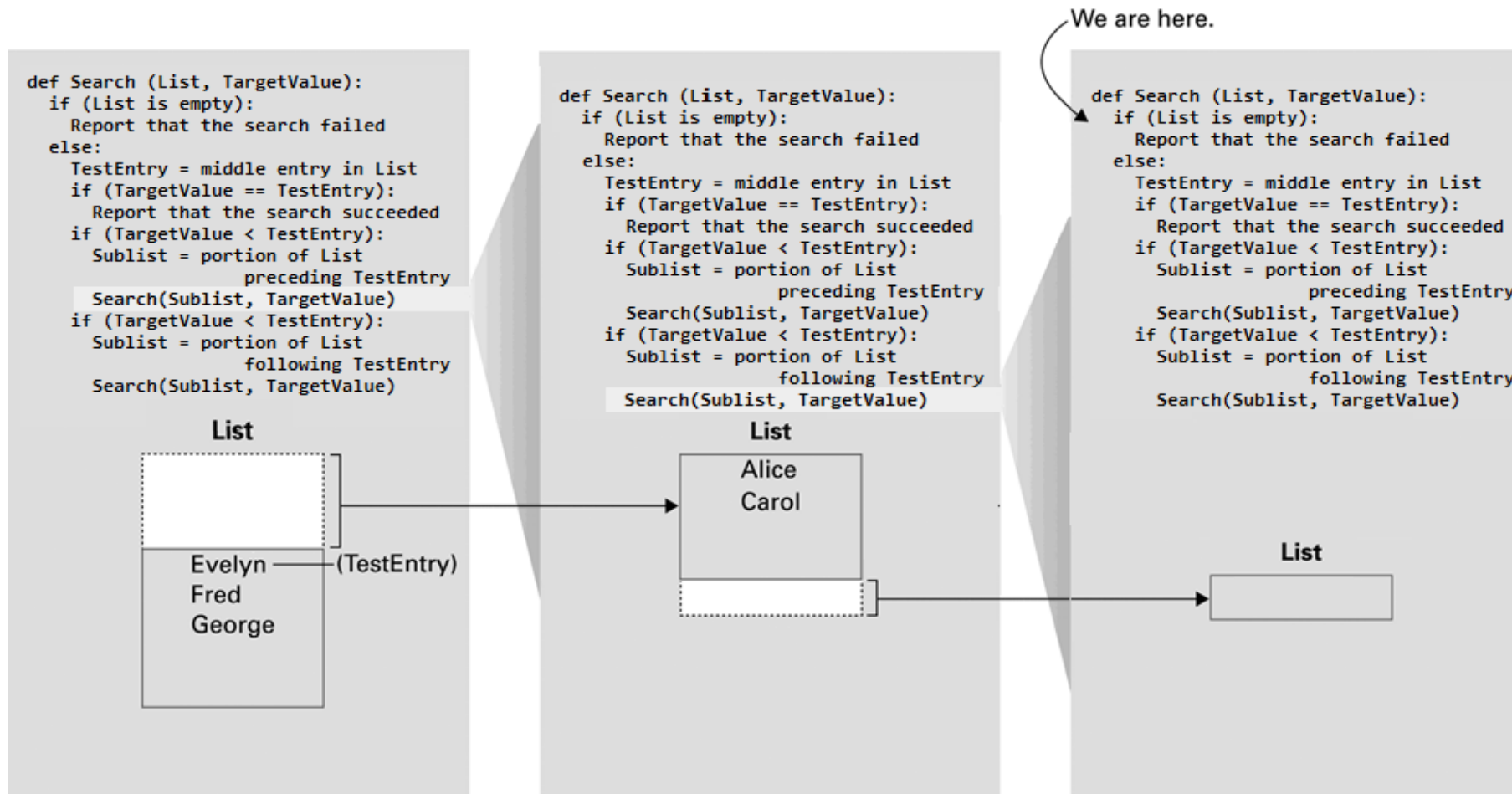
The caller waits for the smaller search. Source correction: the final comparison should use >.

Recursive search: another starting list



Different input gives different sublists. Source correction: the final comparison should use >.

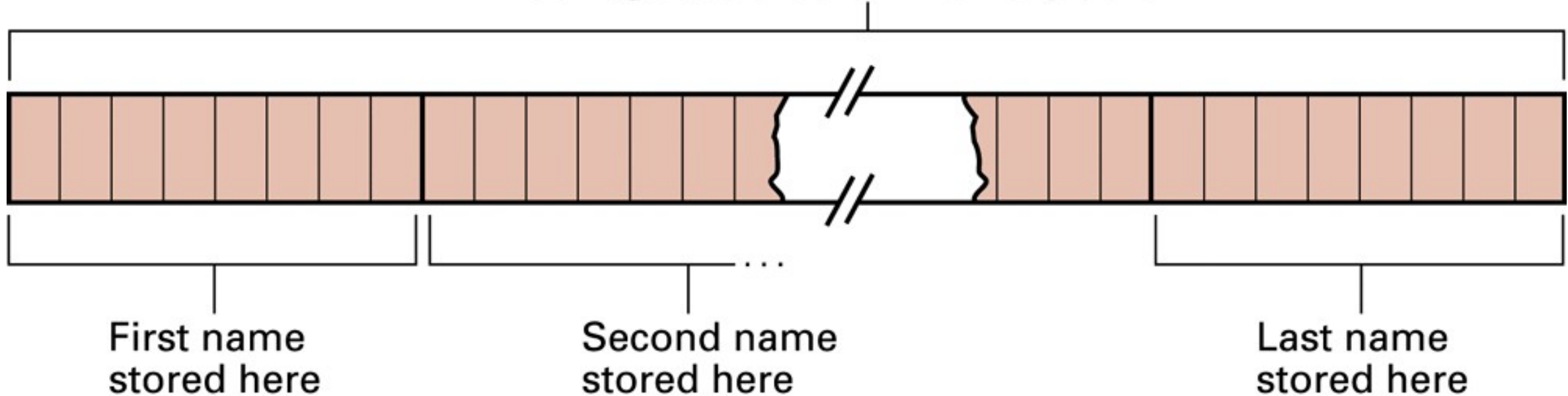
Recursive search: empty subproblem



The empty call reports failure. Source correction: the final comparison should use >.

Names in contiguous storage

Contiguous block of memory cells



Equal-size slots permit indexed access. A slot holding a name can occupy multiple bytes.

Reference: linear search

```
int find(const int a[], int n, int target) {  
    for (int i = 0; i < n; ++i) {  
        if (a[i] == target)  
            return i;  
    }  
    return -1;  
}
```

n is nonnegative; the array
has at least n elements.

First matching index, or -1 .

The body never reads index
 n .

Reference: iterative binary search

```
int binary(const int a[], int n, int t) {  
    int lo = 0, hi = n - 1;  
    while (lo <= hi) {  
        int mid = lo + (hi - lo) / 2;  
        if (a[mid] == t) return mid;  
        if (a[mid] < t) lo = mid + 1;  
        else hi = mid - 1;  
    }  
    return -1;  
}
```

Precondition: ascending
order, valid n and array.

Inclusive bounds: lo...hi.

Any matching index is
acceptable for duplicates.